Implementation and Evaluation of Binary Swap Volume Rendering on a Commodity–Based Visualization Cluster

Xavier Cavin, Mark Hartner and Charles Hansen*

SCI Institute — University of Utah
http://www.sci.utah.edu

Abstract

This paper describes the implementation and performance evaluation of a parallel volume renderer capable of handling large volumetric data sets. We implement our volume renderer on a cluster of 32 Linux PC's using OpenGL, MPI and a binary–swap compositing algorithm. We also give hints for achieving good performance when using OpenGL and MPI on a Linux visualization cluster.

1 Introduction

Volume rendering is a useful and viable method for visualization. Interactive volume rendering is necessary for data exploration and setting of transfer functions. However, as dataset sizes increase, the entire volume does not fit into the texture memory of commodity graphics cards. A cluster based approach that distributes the data provides a scalable solution to this problem. In this paper, we describe the implementation and performance evaluation of binary–swap volume renderer on a commodity–based visualization cluster.

In Section 2, we briefly review the binary–swap volume rendering algorithm. In Section 3, we present the implementation details. We describe in Section 4 the performance evaluation. Finally, we conclude and present future work in Section 5.

2 Binary–Swap Volume Rendering

The binary–swap volume rendering algorithm proceeds as illustrated by Figure 1. Eight nodes are shown in this figure. Looking at a particular node from left to right, the green is the rendering stage, the blue is the framebuffer readback stage. The red/yellow pairs represent the binary–swap stage while the orange is the final image assembly. These steps are outlined below:

1. each node renders its particular subvolume for the current viewpoint (green);

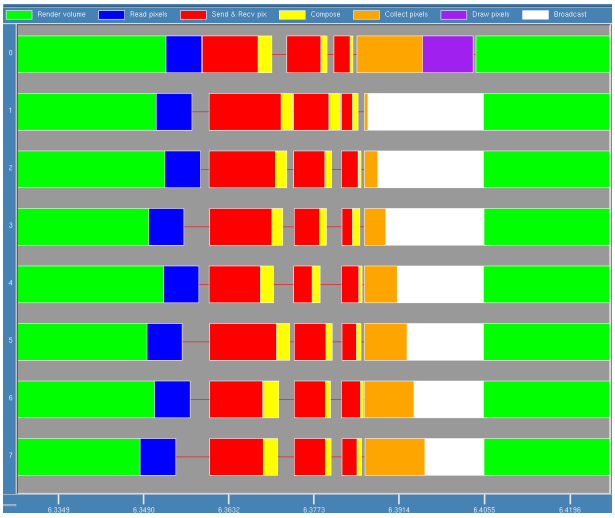

Figure 1: Typical run of the binary-swap volume renderer with 8 nodes (512x512 resolution).

2. the nodes read back the rendered subimages (blue);
3. the nodes exchange subimages (binary–swap) across the network (red) and perform alpha compositing (yellow);
4. the nodes send composited image slices back to the main graphics node (orange);
5. the main graphics node displays the complete image and sends out a new viewpoint (purple).

3 Implementation

3.1 Cluster hardware

Our binary–swap volume renderer has been implemented on a 32–node PC cluster using the OpenGL and MPI libraries.

The main hardware components of each node are shown on Table 1. The 32 nodes are interconnected through a Extreme Networks 6816 BlackDiamond switch with Gigabit ethernet over Copper.

*mailto:hansen@cs.utah.edu

Component	Type
Motherboard	Supermicro P4DC6+ (Intel 860 chipset)
CPU	Dual Intel Xeon 1.70GHz (256KB cache size)
Memory	2x512M Corsair ECC RDRAM
Network card	Intel Pro1000/XT
Graphics board	NVIDIA
GPU	NVIDIA GeForce3 NV20 (64MB RAM)
Hard drive	18GB Seagate Cheetah U160 (15000 RPM)

Table 1: Hardware components of a single node.

Each node runs a Linux kernel 2.4.19 (Aug 5, 2002) and a Xfree86 server 4.2.0 (Jan 14, 2002). Table 2 lists the drivers used for the different hardware components.

Component	Driver
Network	Intel e1000 4.3.2 (June 11, 2002)
Graphics	GLX and Kernel 1.0-2960 (May 14, 2002)
AGP	AGP4x w/kernel AGPGART

Table 2: Drivers used on a single node.

The application is compiled separately with the GCC compiler 3.0.4 or the Intel C++ compiler 6.0. We also have experimented with two different implementations of MPI: MPICH [5] 1.2.4 (May 7, 2002) and LAM/MPI [3] 6.5.6 (?).

The performance evaluation has been performed with the MultiProcessing Environment (MPE) of the Performance Visualization for Parallel Programs project [7], both with MPICH and LAM/MPI.

3.2 Volume rendering

The volume rendering code we use is based on the Linux Version 1.0 Beta distribution of the Simian software [8]. To speed up rendering on the GeForce3, Simian utilizes several textures. For volumes whose respective textures are small enough to fit into the graphic card's memory, Simian can render at interactive rates. A 128x128x256 byte volume can fit into the GeForce3's texture memory (for the given lighting model we use) and can run on a single node at 20 FPS.

For larger datasets, the volume must be broken down into several subvolumes, which are swapped, similar to virtual memory paging, between texture and main memory during rendering. A 256x256x256 byte volume requires four subvolumes, and the framerate drops to about 0.5 FPS when rendering on a single node.

Our parallel volume renderer is designed to efficiently visualize large datasets at interactive rates. To achieve interactive rates, a given dataset is decomposed into 2^n subvolumes, where each subvolume is small enough to fit into texture memory without memory swapping. Each of the 2^n nodes is given one subvolume and is responsible for rendering it with the Simian software. To produce the final image, binary–swap compositing is performed as previously described.

4 Performance Evaluation

We examine separately the different components of the binary–swap volume rendering algorithm described in §2. Indeed, most of these components are independent of the specific rendering algorithm used and could be applied to other rendering methods (*i.e.* polygon rendering). For our system, we use a 32–bit RGBA framebuffer with 8 bits for each color component and 8 bits for the alpha channel.

4.1 Framebuffer read and write

The first key component is the read and write operations on the graphics board's framebuffer. The OpenGL API provides `glReadPixels()` and `glDrawPixels()` functions to read and write rectangular areas of the framebuffer. There can be a large performance penalty for both of these operations if the following steps are not taken (see [2]):

- All `glPixelTransfer()` states should be set to their default values.
- `glPixelStore()` should be set to its default value, with the exception of `GL_PACK_ALIGNMENT` and `GL_UNPACK_ALIGNMENT`, which should be set to 8 (rows start on double–word boundaries).
- The type and format parameters of `glReadPixels()` and `glDrawPixels()` must correspond to the framebuffer configuration, in order to avoid data translation between the parameter specification and the framebuffer format. In our case (32–bit RGBA with 8 bit alpha channel) we use `GL_RGBA` and `GL_UNSIGNED_INT_8_8_8_8`.

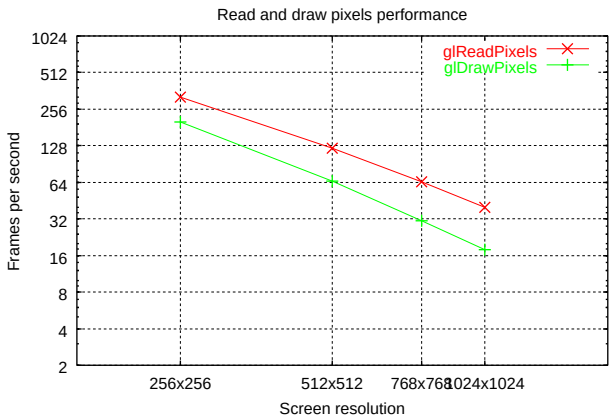

Figure 2: Performance for reading and drawing a 32–bit RGBA framebuffer on a single node.

On a single node of our cluster, we are able to read between 22 and 43 millions pixels per second and to draw between 16 and 19 millions pixels per second, depending on the framebuffer resolution. Figure 2 shows the number of frames that can be read and drawn per second for different resolutions. This gives us a first upper bound of the maximum obtainable framerate.

4.2 Binary–swap and collect

The binary–swap algorithm [4] requires 2^n nodes and proceeds in n stages by splitting the image at each stage, with pairs of nodes operating on different subimages. At the end of the process, the image is partitioned among all of the nodes, requiring a final image assembly step to retrieve all of the pieces.

Binary–swap stages

At each stage of the binary–swap, pairs of nodes exchange one half of the subimage they have to compose. This exchange is implemented using the MPI `MPI_Sendrecv()` function. This operation combines in one call the sending of a message to one destination and the receiving of another message from another node; the two (source and destination) are the same in our case.

For our particular communication pattern, we found LAM/MPI to be much faster and much more reliable than MPICH. Using MPICH, we found that occasionally a given `MPI_Sendrecv()` call between node pairs took four times longer than equivalent exchanges among other node pairs. We have found no particular explanation to this pheonenum, but the efficiency of LAM/MPI for `MPI_Sendrecv()` has also been described by the CLIC project [1].

We also found that knowing the appropriate options to LAM/MPI `mpirun` can produce much better network performance. When running on a homogeneous cluster such as ours, the `-O` option to `mpirun` eliminates data conversion when passing messages, and the network performance improves between 25% and 100%, depending on the number of nodes and message sizes.

Final collecting step

During the final collecting step, each node sends (`MPI_Send()`) $1/2^n$ of the image to a master node, that collects (`MPI_Recv()`) these 2^n (or $2^n - 1$) subimages one after the other before rendering the final image. In an ideal world with a zero–latency communication, the time needed for this step is directly proportional to the size of the final image and is independent of the number of nodes. In practice, the overhead of the final collecting step is more important for smaller resolutions.

Figure 3 shows the number of frames per second the binary–swap algorithm can handle for different screen resolutions (excluding and including the final collecting step). This gives us another upper bound of the final obtainable framerate.

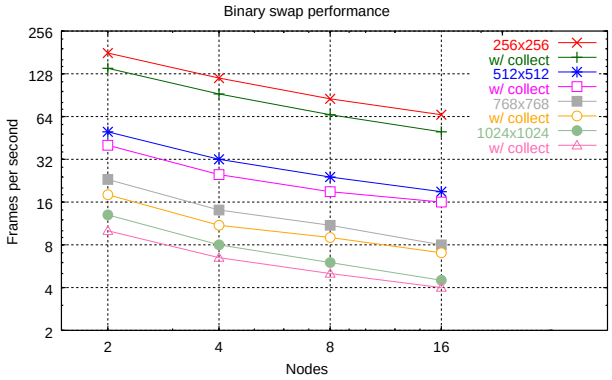

Figure 3: Performance of the binary-swap algorithm on our cluster with LAM/MPI (with and without final collecting step).

4.3 Image compositing

The image compositing algorithm is a loop over two arrays of pixels. In general, each pixel is composed of three color components and an alpha channel. During the loop, each pair of pixels is blended with respect to their alpha value and relative position.

Image compositing is an embarrassingly parallel algorithm. Indeed for each pair of pixels, blending is independent of the others and parallelism can be exploited at the thread level and at the instruction set level (for instance with vectorized operations).

In our case, image compositing is not the bottleneck (see Figure 1), so we have not focused on optimizing it. However, since our cluster nodes have dual processors, we have parallelized the compositing loop, by adding *one single* OpenMP [6] compilation directive (`#pragma omp parallel for`). Thus, when we use a compiler that supports OpenMP (as does the Intel C++ Compiler), the code is executed in parallel, and the compositing can easily be accelerated. We saw an acceleration by a factor of two.

4.4 Overall performance

As an experiment, we tested our binary–swap volume rendering system on a 512x512x209 byte dataset from the head section of the Visible Woman CT data (see Figure 4), with 4, 8 and 16 of our cluster nodes and different resolutions from 256x256 to 1024x1024[1]. Table 3 shows the obtained framerates.

[1]We were not able to experiment with the 32 nodes of the cluster due to current hardware problems with our main switch.

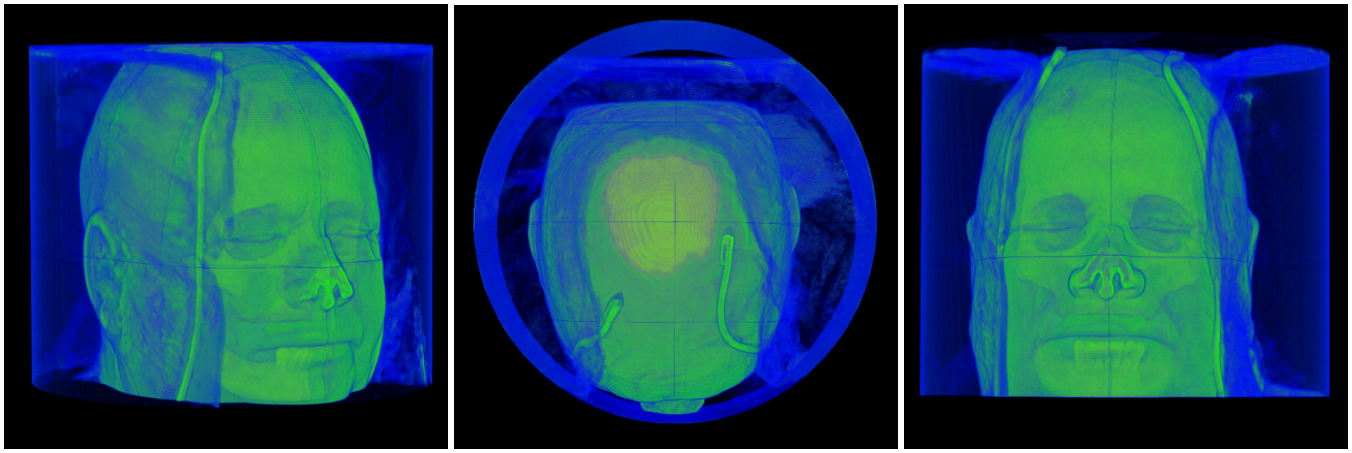

Figure 4: Views of the head section (512x512x209) of the visible female CT data with 16 nodes. We have left a small space between the subvolumes to highlight their respective borders.

Nodes	256x256	512x512	768x768	1024x1024
4	0.3	0.3	0.3	0.3
8	38	12	6	3.6
16	44	13	6.5	3.7

Table 3: Framerates with different resolutions and different numbers of nodes.

The obtained framerates can be explained by the texture swapping occuring with subvolumes larger than 128x128x256. With 4 nodes, each subvolume is 256x256x256: the rendering performance on a single node is dramatically slow. With 16 nodes, each subvolume is 128x128x256, which is the optimal size for rendering on a single node. With 8 nodes, each subvolume is 128x256x256 and induces texture swapping: the rendering time for a particular node is higher than in the 16 nodes configuration. However, as shown by Figure 3, the binary–swap performance is better with 8 nodes than with 16 nodes: the total image generation time for a single frame is then equivalent for both.

5 Conclusion and Future Work

This paper describes preliminary work on utilizing a PC graphics cluster based on commodity parts for volume visualization. We have described practical aspects of our system which should be of interest to others performing similar work. In the future, we would like to exploit faster rendering methods to speedup our system. We would also like to experiment with the SSE (vectorizing) instructions found on the Pentium4 CPU. Another area of research is the use of multi–resolution based volume rendering to further increase the size of volumes rendered on a particular system. We believe that cluster–based volume rendering will be used extensively for large–scale visualization in the future.

Acknowledgments

We would like to thank Joe Kniss for providing the volume rendering code on which this work is based, and Gordon Kindlmann for providing his teem software [9] to manipulate volumetric datasets. This material is based upon work supported by the AVTC under the DOE VIEWS program.

References

[1] *CLIC: Chemnitzer Linux Cluster*. http://www-user.tu-chemnitz.de/~pester/CLIC.

[2] *OpenGL Developer FAQ and TroubleShooting Guide*. http://www.opengl.org/developers/faqs/technical.html.

[3] *LAM/MPI Parallel Computing*. http://www.lam-mpi.org.

[4] Kwan-Liu Ma, James Painter, Charles Hansen, and Michael Krogh. Parallel volume rendering using binary–swap compositing. *IEEE Computer Graphics and Applications*, 14(4):59–68, 1994.

[5] *MPICH — A Portable Implementation of MPI*. http://www-unix.mcs.anl.gov/mpi/mpich.

[6] *OpenMP*. http://www.openmp.org.

[7] *Performance Visualization for Parallel Programs*. http://www-unix.mcs.anl.gov/perfvis.

[8] *Simian*. http://www.cs.utah.edu/~jmk/simian.

[9] *teem: GK's C code*. http://www.cs.utah.edu/~gk/teem.